## **Development Plan for Linka E-Commerce Platform**

### **Project Overview**

Linka is an e-commerce platform for SMEs in Zambia, connecting customers, shops, and delivery systems via a proxy architecture. The development plan focuses on:

1. Setting up Google Cloud environments for the proxy server.
2. Developing the shop-application in a Docker container.
3. Connecting the shop-application to the proxy.
4. Building the client-application (user app).
5. Integrating the client-application with the proxy.

### **Timeline**

Total estimated duration: **4 weeks.**

### **Phase 1: Set Up Google Cloud Environments for Proxy Server**

**Duration**: 2 Days (Week 1, Day 1-2)  
 **Objective**: Provision and configure the proxy server environment on Google Cloud Compute Engine (GCE) with PostgreSQL for data persistence.

#### **Steps**

1. **Create a Google Cloud Project**
   1. Sign into Google Cloud Console.
   2. Create a new project: linka-ecommerce-prod.
   3. Enable billing and APIs: Compute Engine API, Cloud SQL API.
2. **Provision Compute Engine Instance**
   1. Create an e2-micro instance:
      1. **Region**: us-central1 (or closest available to Zambia, e.g., europe-west4 for lower latency).
      2. **Machine Type**: e2-micro (2 vCPUs, 1 GB RAM).
      3. **Boot Disk**: 10 GB SSD, Ubuntu 22.04 LTS.
      4. **Firewall**: Allow HTTP/HTTPS (ports 80, 443).
   2. Set up SSH access for the development team.
3. **Set Up PostgreSQL with Cloud SQL**
   1. Create a PostgreSQL instance:
      1. Version: PostgreSQL 16.
      2. Tier: db-f1-micro (1 vCPU, 614 MB RAM) for development.
      3. Storage: 10 GB SSD.
      4. Enable private IP for secure access from GCE.
   2. Install PostGIS extension: CREATE EXTENSION postgis;.
   3. Create database: linka\_db.
   4. Schema setup:

sql

CollapseWrap

Copy

**CREATE** **TABLE** shops ( shop\_id VARCHAR(36) **PRIMARY** KEY, name VARCHAR(100), location GEOGRAPHY(POINT), verified BOOLEAN **DEFAULT** FALSE, created\_at TIMESTAMP **DEFAULT** CURRENT\_TIMESTAMP, updated\_at TIMESTAMP **DEFAULT** CURRENT\_TIMESTAMP ); **CREATE** INDEX idx\_shops\_location **ON** shops **USING** GIST (location);

1. **Install Dependencies on GCE Instance**
   1. SSH into the instance.
   2. Update system: sudo apt update && sudo apt upgrade -y.
   3. Install Python: sudo apt install python3.12 python3-pip -y.
   4. Install FastAPI and dependencies: pip install fastapi uvicorn psycopg2-binary.
   5. Install gcloud CLI for local testing.
2. **Deploy a Basic Proxy Server**
   1. Create a minimal FastAPI app:

python

CollapseWrapRun

Copy

**from** fastapi **import** FastAPI app = FastAPI() @app.get("/health") **async** **def** **health\_check**(): **return** {"status": "healthy"}

* 1. Run the app: uvicorn main:app --host 0.0.0.0 --port 80.
  2. Test: curl http://<instance-ip>/health.

#### **Deliverables**

* Google Cloud project with GCE instance and Cloud SQL (PostgreSQL) set up.
* Basic FastAPI proxy server running with a /health endpoint.

### **Phase 2: Create the Shop-Application in Docker**

**Duration**: 3 Days   
 **Objective**: Build the shop-application as a Docker container to manage shop operations (catalog, orders, payment forms).

#### **Steps**

1. **Set Up Development Environment** (2 Hours)
   1. Install Docker on a local machine: sudo apt install docker.io -y.
   2. Verify: docker --version (should be Docker 20.x or higher).
   3. Install Node.js (v20.x): curl -fsSL <https://deb.nodesource.com/setup_20.x> | sudo -E bash - && sudo apt install -y nodejs.
2. **Develop Shop-Application** (8 Hours)
   1. Create project structure:

text

CollapseWrap

Copy

shop-app/ ├── src/ │ ├── index.js │ └── routes/ │ └── catalog.js ├── Dockerfile ├── package.json └── .env

* 1. Set up Express.js app (src/index.js):

javascript

CollapseWrapRun

Copy

**const** express = require('express'); **const** mongoose = require('mongoose'); **const** catalogRoutes = require('./routes/catalog'); **const** app = express(); app.use(express.json()); mongoose.connect(process.env.MONGO\_URI, { useNewUrlParser: true, useUnifiedTopology: true }); app.use('/catalog', catalogRoutes); app.listen(3000, () => console.log('Shop app running on port 3000'));

* 1. Define catalog route (src/routes/catalog.js):

javascript

CollapseWrapRun

Copy

**const** express = require('express'); **const** router = express.Router(); router.get('/', (req, res) => { // Mock catalog data res.json([ { id: 'p1', name: 'Bread', price: 2.0, stock: 10 }, { id: 'p2', name: 'Milk', price: 3.0, stock: 5 } ]); }); module.exports = router;

* 1. Configure package.json:

json

CollapseWrap

Copy

{ "name": "shop-app", "version": "1.0.0", "dependencies": { "express": "^4.18.2", "mongoose": "^8.0.0" }, "scripts": { "start": "node src/index.js" } }

1. **Containerize the Application** (4 Hours)
   1. Create Dockerfile:

dockerfile

CollapseWrap

Copy

**FROM** node:20-alpine **WORKDIR** /app **COPY** package.json ./ **RUN** npm install **COPY** . . **EXPOSE** 3000 **CMD** ["npm", "start"]

* 1. Build and test locally:

bash

CollapseWrapRun

Copy

docker build -t shop-app:latest . docker run -d -p 3000:3000 --env MONGO\_URI=mongodb://host.docker.internal:27017/shop\_db shop-app:latest

* 1. Test: curl <http://localhost:3000/catalog>.

#### **Deliverables**

* Dockerized shop-application with /catalog endpoint.
* Local MongoDB instance for development (assumed running on host).

### **Phase 3: Connect Shop-Application to Proxy**

**Duration**: 2 Days   
 **Objective**: Enable the proxy to communicate with the shop-application for catalog retrieval and order processing.

#### **Steps**

1. **Deploy Shop-Application on Google Cloud** (4 Hours)
   1. Push Docker image to Google Container Registry (GCR):

bash

CollapseWrapRun

Copy

docker tag shop-app:latest gcr.io/linka-ecommerce-prod/shop-app:latest docker push gcr.io/linka-ecommerce-prod/shop-app:latest

* 1. Deploy on GCE or Cloud Run:
     1. Use Cloud Run for simplicity: gcloud run deploy shop-app --image gcr.io/linka-ecommerce-prod/shop-app:latest --region us-central1 --port 3000.
     2. Note the service URL (e.g., <https://shop-app-xyz.run.app>).

1. **Update Proxy Server** (4 Hours)
   1. Install HTTP client: pip install httpx.
   2. Add shop discovery and catalog relay endpoints (main.py):

python

CollapseWrapRun

Copy

**from** fastapi **import** FastAPI **import** httpx app = FastAPI() SHOP\_APP\_URL = "<https://shop-app-xyz.run.app>" @app.post("/shops/discover") **async** **def** **discover\_shops**(data: dict): lat, long = data["lat"], data["long"] # Query PostgreSQL with PostGIS # For now, mock a shop with ID and URL **return** [{"shop\_id": "s1", "url": SHOP\_APP\_URL}] @app.get("/shops/{shop\_id}/catalog") **async** **def** **get\_catalog**(shop\_id: str): **async** **with** httpx.AsyncClient() **as** client: response = **await** client.get(f"{SHOP\_APP\_URL}/catalog") **return** response.json()

* 1. Redeploy proxy: gcloud compute instances update linka-proxy --zone us-central1-a --restart.

1. **Test Connectivity** (2 Hours)
   1. Test shop discovery: curl -X POST http://<proxy-ip>/shops/discover -d '{"lat": -15.3875, "long": 28.3228}'.
   2. Test catalog retrieval: curl http://<proxy-ip>/shops/s1/catalog.

#### **Deliverables**

* Shop-application deployed on Cloud Run.
* Proxy server updated to relay requests to shop-application.

### **Phase 4: Develop the Client-Application**

**Duration**: 5 Days (Week 2, Day 3 - Week 3, Day 2)  
 **Objective**: Build a React.js-based client-application for users to browse shops and place orders.

#### **Steps**

1. **Set Up React Project** (4 Hours)
   1. Initialize project with Vite:

bash

CollapseWrapRun

Copy

npm create vite@latest client-app -- --template react cd client-app npm install

* 1. Install dependencies: npm install axios tailwindcss postcss autoprefixer.
  2. Set up Tailwind CSS:

bash

CollapseWrapRun

Copy

npx tailwindcss init -p

Update tailwind.config.js:

javascript

CollapseWrapRun

Copy

module.exports = { content: ["./index.html", "./src/\*\*/\*.{js,jsx}"], theme: { extend: {} }, plugins: [], };

Add to src/index.css:

css

CollapseWrap

Copy

**@tailwind** base; **@tailwind** components; **@tailwind** utilities;

1. **Develop Core Features** (12 Hours)
   1. Create components for shop discovery and catalog browsing:
      1. src/App.jsx:

javascript

CollapseWrapRun

Copy

**import** { useState, useEffect } **from** 'react'; **import** axios **from** 'axios'; **function** **App**() { **const** [shops, setShops] = useState([]); **const** [catalog, setCatalog] = useState([]); **const** PROXY\_URL = 'http://<proxy-ip>'; useEffect(() => { // Get user location (mocked for now) **const** location = { lat: -15.3875, long: 28.3228 }; axios.post(`${PROXY\_URL}/shops/discover`, location) .then(res => setShops(res.data)); }, []); **const** fetchCatalog = (shopId) => { axios.get(`${PROXY\_URL}/shops/${shopId}/catalog`) .then(res => setCatalog(res.data)); }; **return** ( <**div** className="p-4"> <**h1** className="text-2xl">Linka E-Commerce</**h1**> <**h2**>Shops Near You</**h2**> {shops.map(shop => ( <**div** key={shop.shop\_id}> <**button** onClick={() => fetchCatalog(shop.shop\_id)}> View {shop.shop\_id} Catalog </**button**> </**div**> ))} <**h2**>Catalog</**h2**> <**ul**> {catalog.map(item => ( <**li** key={item.id}>{item.name} - ${item.price}</**li**> ))} </**ul**> </**div**> ); } **export** **default** App;

* 1. Test locally: npm run dev.

1. **Optimize and Build** (4 Hours)
   1. Add error handling and loading states to components.
   2. Build for production: npm run build.

#### **Deliverables**

* React.js client-application with shop discovery and catalog browsing features.
* Local development server for testing.

### **Phase 5: Connect Client-Application to Proxy**

**Duration**: 3 Days (Week 3, Day 3 - Week 4, Day 1)  
 **Objective**: Integrate the client-application with the proxy server to enable full user interaction.

#### **Steps**

1. **Deploy Client-Application** (4 Hours)
   1. Deploy to Netlify:

bash

CollapseWrapRun

Copy

npm install -g netlify-cli netlify deploy --prod

* 1. Set environment variable for proxy URL in Netlify dashboard: PROXY\_URL=http://<proxy-ip>.

1. **Enhance Proxy Server for Client Interactions** (6 Hours)
   1. Add order placement endpoint:

python

CollapseWrapRun

Copy

@app.post("/order") **async** **def** **place\_order**(data: dict): shop\_url = data["shop\_url"] **async** **with** httpx.AsyncClient() **as** client: response = **await** client.post(f"{shop\_url}/order", json=data) **return** response.json()

* 1. Redeploy proxy.

1. **Update Client-Application for Ordering** (4 Hours)
   1. Add order functionality to App.jsx:

javascript

CollapseWrapRun

Copy

**const** placeOrder = **async** () => { **const** order = { shop\_url: shops[0].url, items: [{ id: 'p1', quantity: 2 }], }; **await** axios.post(`${PROXY\_URL}/order`, order); alert('Order placed!'); };

* 1. Add a button: <button onClick={placeOrder}>Place Order</button>.

1. **Test End-to-End** (2 Hours)
   1. Test flow: Discover shops → View catalog → Place order.
   2. Verify proxy relays requests correctly.

#### **Deliverables**

* Client-application deployed on Netlify.
* Proxy updated with order placement endpoint.
* End-to-end flow tested successfully.

### **Phase 6: Final Testing and Documentation**

**Duration**: 2 Days (Week 4, Day 2-3)  
 **Objective**: Ensure system stability and document setup for future reference.

#### **Steps**

1. **Final Testing** (6 Hours)
   1. Load test proxy with JMeter: 100 concurrent users.
   2. Verify shop-application scalability in Cloud Run.
   3. Test client-application on multiple devices (desktop, mobile).
2. **Documentation** (2 Hours)
   1. Document setup steps for Google Cloud, Docker, and deployment processes.
   2. Create API documentation with Swagger for proxy endpoints.

#### **Deliverables**

* Stable, tested system.
* Comprehensive setup and API documentation.

### **Resource Requirements**

* **Team**: 1 DevOps engineer (Google Cloud setup), 2 backend developers (proxy, shop-app), 1 frontend developer (client-app).
* **Tools**:
  + Google Cloud Console, Docker, Node.js, React.js, Netlify CLI.
  + JMeter for load testing, Swagger for API docs.